42P17256

UNITED STATES PATENT APPLICATION

FOR

Method of Obscuring Cryptographic Computations

INVENTOR:

Ernie F. Brickell

INTEL CORPORATION

Prepared by:

Steven P. Skabrat

Reg. No. 36,279

(503) 264-8074

Express Mail No. EV 325530002 US

Method of Obscuring Cryptographic Computations

BACKGROUND

1. FIELD

The present invention relates generally to cryptography and, more specifically, to deterring attacks based at least in part on observing cryptographic computations in a processing system.

2. DESCRIPTION

Public key cryptography is well-known and widely used today. In public key cryptography, each user has a public key and a private key. The public key is made public, while the private key remains secret. Encryption is performed with the public key, while decryption is done with the private key.

The most popular form of public key cryptography today is the Rivest, Shamir, Adleman (RSA) public key cryptosystem. Key generation in the RSA algorithm works as follows: take two large primes, **p** and **q**, and compute their product **n=p*q**; **n** is called the modulus. Choose a number, called the encryption exponent **e**, less than **n** and relatively prime to (**p**-1)*(**q**-1), which means **e** and (**p**-1)*(**q**-1) have no common factors except 1. Find another number, called the decryption exponent **d**, such that (**e*d**-1) is divisible by (**p**-1)*(**q**-1). The values **e** and **d** are also called the public and private exponents, respectively. The public key is the pair (**n**, **e**) and the private key is the exponent d. The factors **p** and **q** may be destroyed or kept with the private key.

Encryption and decryption may be performed as follows. Suppose Alice wants to send a plaintext message **m** ($0 \le \mathbf{m} \le \mathbf{n}-1$) to Bob. Alice creates the ciphertext message **c** ($0 \le \mathbf{c} \le \mathbf{n}-1$) by exponentiating $\mathbf{c} = \mathbf{m}^{\mathbf{e}} \bmod \mathbf{n}$, where **e** and **n** are Bob's public key. She sends **c** to Bob. To decrypt, Bob exponentiates $\mathbf{m} = \mathbf{c}^{\mathbf{d}} \bmod \mathbf{n}$; the relationship between **e** and **d** ensures that Bob correctly recovers **m**. Since only Bob knows **d**, only Bob can decrypt this message.

RSA is also commonly used to exchange a symmetric key. A symmetric key may be used to encrypt data. The symmetric key may then be encrypted by a sender using the receiver's public key. In this usage, the message **m** is a specific formatting of the symmetric key. Once the receiver receives and decrypts the symmetric key, the receiver can then use the symmetric key to decrypt the data.

It is currently very difficult to obtain a private key **d** from the public key (**n**, **e**). However, if one could factor **n** into **p** and **q**, then one could obtain the private key **d**. Thus, the security of the RSA system is based on the assumption that such factoring is difficult.

Other attacks on the RSA system have been attempted. In some sophisticated attack scenarios, **d** may inferred from information gathered from observing a processing system performing the modular exponentiation operation used in decryption. In these scenarios, observing memory access patterns, cache line accesses, and/or branches taken in executing code within the processing system may give the attacker sufficient information to deduce the private key. Hence, obscuring cryptographic computations such that observation of the memory access patterns, cache line accesses, and/or branches taken in executing code while performing the computations provides no meaningful information to an attacker is desirable.

BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the present invention will become apparent from the following detailed description of the present invention in which:

Figure 1 is a flow diagram illustrating obscuring cryptographic computations according to an embodiment of the present invention; and

Figure 2 is a flow diagram illustrating obscuring cryptographic computations according to another embodiment of the present invention.

DETAILED DESCRIPTION

An embodiment of the present invention is a method for implementing a variant of the RSA algorithm in a secure manner to deter against attacks by obscuring cryptographic computations such that an attacker can gain no meaningful information by observing, for example, memory access patterns, cache line accesses, or code branches of a processing system. Embodiments of the present invention provide a method of performing an implementation of the RSA algorithm such that memory access is independent of the bit pattern of the exponents used. Embodiments of the present invention present a new way to protect against timing attacks. Embodiments of the present invention are also more efficient than previous methods.

Reference in the specification to "one embodiment" or "an embodiment" of the present invention means that a particular feature, structure or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrase "in one embodiment" appearing in various places throughout the specification are not necessarily all referring to the same embodiment.

To decrypt a ciphertext message **c** using the RSA system to obtain the plaintext message **m**, operations are performed to compute $\mathbf{c}^{\mathbf{d}}$ mod **n**. An attacker may be able obtain **c** by capturing communications between a sender and a receiver, and **n** is known from the public key, but the difficulty in making the attack is in determining **d**. If an attacker can monitor or otherwise observe a processing system during a decryption operation, the attacker may be able to infer or deduce what **d** is. Once **d** is known, the attacker can decrypt any messages encrypted with the sender's corresponding public key. In some scenarios, observing memory access patterns, cache line accesses, and/or branches taken or not taken in executing code may help the attacker obtain **d**. In some instances, this observation may be made using a logic analyzer on a processing system performing the decryption operation.

One known implementation of the RSA algorithm for decryption is shown in Table 1. $\mathbf{d}_i$ will denote the i'th bit of **d**. Specifically, $d = \sum_{i=0}^{k-1} d_i 2^i$, where k is the number of bits in the binary representation of d. In other words, $2^{k-1} \leq d < 2^k$.

Table I

```
1: Let k = log₂ n      /* k is the number of bits in d */
2: x: = c                        /* store ciphertext c in intermediate value variable */
3: For i = 2 to k                /* loop through all bits in d */
4:     x:= x * x MOD n
5:     If d_{k-i} = 1,     /* if i'th most significant bit of d is set     */
6:            then x := x * c mod n
7: End For
8: m:= x      /* store recovered plaintext m  */
```

The well-known implementation shown in Table I may be vulnerable to observations by an attacker because the memory reference to **c** in line 6 only occurs when the i'th bit of decryption exponent **d** is set to 1. When the i'th bit of decryption exponent **d** is 0, no memory access is made. If an attacker observes the memory access activity during decryption processing, the attacker may obtain the bit pattern for **d**. Furthermore, monitoring of whether the branch is taken at line 5 may provide the same information. Once the attacker learns **d**, the attacker can obtain **m**.

To avoid this scenario, an embodiment of the present invention performs decryption processing wherein memory access is independent of the decryption exponent bit pattern. Table II shows an embodiment of the present invention wherein lines 5 and 6 of the prior art method may be replaced by processing designed to obfuscate decryption computations.

Table II

```
1: Let $\mathbf{k} = \log_2 \mathbf{n}$     /* k is the number of bits in d */
2: x: = c                  /* store ciphertext c in intermediate value variable */
3: For I – 2 to k          /* loop through all bits in d */
4:     x := x * x MOD n
5:     r := (c -1) * $\mathbf{d}_{k-I}$ + 1   /* determine obscuring factor */
6:     x := x * r MOD n    /* use obscuring factor */
7: End For
8: m := x                  /* store plaintext m */
```

In this embodiment, an obscuring factor **r** is used. By using this obscuring factor in the multiplication at line 6, there is no conditional memory access or branch pattern. Each iteration **i** through the bits of decryption exponent **d** involves a memory access to obtain **c** in line 5 and a multiplication in line 6, regardless of the bit pattern of **d**. Thus, no pattern may be observed by an attacker. When $\mathbf{d}_i = 0$, then $\mathbf{r} = 1$ in line 5, and a multiplication by 1 is performed in line 6. When $\mathbf{d}_i = 1$, then $\mathbf{r} = \mathbf{c}$ in line 5, and a multiplication by **c** is performed in line 6. Furthermore, there is no branch activity (from evaluating an "if" statement) that can be observed to deduce **d**. At the end of processing in Table II, intermediate variable **x** stores the same value as at the end of processing in Table I (that is, the ciphertext **c** has been decrypted into plaintext **m**).

Figure 1 is a flow diagram illustrating obscuring cryptographic computations according to an embodiment of the present invention. In general terms, lines 3 through 6 of Table II are represented as a flow diagram. At block 100, as part of decryption processing, a loop may be set up to iterate through each bit **i** in the decryption exponent **d**. At block 102, an intermediate value (initially storing ciphertext **c**) may be set to the intermediate value multiplied by the intermediate value mod **n**. At block 104, the current obscuring factor **r** may be determined using the **i**'th bit of decryption exponent **d**. In one embodiment, the obscuring factor r may be set to ($\mathbf{c}$ – 1) * $\mathbf{d}_i$ + 1. At block 106, the intermediate value may be set to the intermediate value multiplied by the current obscuring factor **r** mod **n**. At block 108, if there are more bits in **d** to process, the next **i** is selected and processing loops back to block 102 for the next iteration of the loop. If all bits of **d** have been processed, then processing ends at block 110.

Thus, a conditional multiplication operation during decryption processing has been replaced with an unconditional multiplication operation such that the end result is equivalent. The method thus performs the decryption operation where memory accesses are independent of the decryption exponent bit pattern and there are no branches to be observed.

In some computing systems, the time to perform a multiply operation by one or by zero may be different than the time to perform a multiply operation by a random number (such as **c**, for example). An attacker may be able to detect this difference in multiplication times to infer the decryption exponent. Another embodiment of the present invention obscures this evidence by changing the way the obscuring factor may be determined. In this embodiment, an arithmetic selection of **r** is performed by using logical operations to avoid a multiplication by $\mathbf{d}_i$ = 0 that could occur in line 5 of Table II. This embodiment avoids the multiplications by one that could occur in line 6 of Table II by always multiplying by a random obscuring factor. Table III shows an embodiment of the present invention wherein the method of Table II may be modified to further obfuscate decryption computations.

Table III

```
1: Let $\mathbf{k} = \log_2 \mathbf{n}$            /* k is the number of bits in d */
2: Pick t at random MOD n                 /* random number t */
3: $y := \mathbf{c} * \mathbf{t}$ MOD n            /* store ciphertext c in intermediate value y using t */
4: $w := \mathbf{c} * \mathbf{t}^{-1}$ MOD n         /* determine first obscuring factor w */
5: $s := \mathbf{t}^{-1}$ MOD n             /* determine second obscuring factor s */
6: For i = 0 to k-1             /* loop through all bits of d */
7:    $y := y * y$ MOD n
8:    $r :=$ (w AND $\mathbf{d}_i$) OR (s AND (NOT $\mathbf{d}_i$))  /* determine third obscuring factor */
9:    $y := r$ MOD n              /* use third obscuring factor r */
10: End For
11: $\mathbf{m} := y * s$ MOD n  /* compute plaintext m using second obscuring factor */
```

In the embodiment shown in Table III, a random number **t** is picked, wherein **t** is between 1 and **n** – 1. The third obscuring factor **r** will evaluate to either the first obscuring factor **w** (when $\mathbf{d}_i = 1$) or the second obscuring factor **s** (when $\mathbf{d}_i = 0$). This embodiment eliminates the multiplication by one inherent in the embodiment of Table II. It is unlikely that either **w** or **s** will be zero or one, considering that **t** is picked at random from a large number set (1 .. **n** – 1). In some embodiments, n may comprise 2^{1024} bits. Note that at each iteration of the For loop, the y calculated in this embodiment will satisfy $y = x * \mathbf{t} \bmod \mathbf{n}$, where x is the value calculated in a given iteration i of the embodiment shown in Table II.

Figure 2 is a flow diagram illustrating obscuring cryptographic computations according to an embodiment of the present invention. In general terms, lines 2 through 11 of Table III are represented as a flow diagram. At block 200, as part of decryption processing, a random number **t** may be picked in the range 1 to **n** – 1. At block 202, an intermediate value may be determined using the plaintext message **m**, random number **t**, and the modulus **n**. At block 204, first and second obscuring factors may be determined. The first obscuring factor w may be set to the plaintext **m** * random number **t** mod **n**. The second obscuring factor s may be set to the inverse of the random number **t** mod **n**. A loop may be set up to iterate through each bit **i** in the decryption exponent **d**. At block 208, the intermediate value determined at block 202 may be set to the intermediate value multiplied by the intermediate value mod **n**. At block 210, the third obscuring factor r may be determined using the **i**'th bit of decryption exponent **d**, the first obscuring factor w, and the second obscuring factor **s**. In one embodiment, the third obscuring factor r may be set to (w AND $\mathbf{d}_i$) OR (**s** AND (NOT $\mathbf{d}_i$)). At block 212, the intermediate value may be set to the intermediate value multiplied by the current third obscuring factor r mod **n**. At block 214, if there are more bits in **d** to process, the next **i** is selected and processing loops back to block 208 for the next iteration of the loop. If all bits of **d** have been processed, then processing continues with block 216. At block 216, plaintext **m** may be obtained by computing the intermediate value multiplied by the second obscuring factor mod **n**.

Thus, in this embodiment, a conditional multiplication operation during decryption processing has also been replaced with an unconditional multiplication operation such that the end result is equivalent. The method performs the decryption operation (in this example) where memory accesses are independent of the decryption exponent bit pattern and there are no branches to be observed. However, in this embodiment, no pattern of multiplying by ones or zeros may be detected.

One known method of computing modular exponentiation is called the window method (also known as the m-ary sliding window method). In the window method, the exponent (such as the decryption exponent **d** or the encryption exponent **e**) may be divided into blocks of bits, where v is the number of bits per block. Each block is a "window" into a portion of the exponent. The window method employs a pre-computation of the powers of the message (such as the ciphertext **c** for decryption or the plaintext **m** for encryption) from 1 to $2^v - 1$. Then, for each bit in the exponent, a squaring operation is performed. Finally, for each window in the exponent, a multiply operation is performed using the appropriate pre-computed power of the message corresponding to the current window and the current window's bits. The appropriate pre-computed power of the message is typically accessed from memory prior to performing the multiply operation.

When using the embodiment of the present invention shown in Table II as applied to the window method, a multiply operation may be performed if the window into the exponent **d** has a value that is not all zeroes. In order to obfuscate the memory access when obtaining the appropriate pre-computed power of the message (e.g., **c**), in an embodiment of the present invention all of the powers of the message from 1 to 2^v may be obtained from memory for each multiply operation even though only one of the powers is actually used in a given iteration. By obtaining all powers for each multiply operation, the memory accesses are independent of which of the powers is being used in a particular multiply operation. Hence, this deters an attacker from obtaining information about the message which may be used to deduce the decryption exponent **d**. The appropriate power of the message may be selected according to $\mathbf{c}^{\mathbf{a}}$ where **a** is the value of the current window into the exponent. This is illustrated in Table IV.

Table IV

© 2003 Intel Corporation

```
1: v: = the number of bits in the window.
2: $c_0 = 1$
3: For i = 1 to $2^v$-1          /* loop through each integer less than $2^v$ */
4:     $c_i = c_{i-1} * c$   MOD **n** /* compute $c_i = c^i$ MOD n */
5: End for
6: **k**: = $\log_2$ **n**        /* **k** is the number of bits in **d** */
```

```
7: Compute $k_v$ and $r_v$ such that k-1 = v*$k_v$ + $r_v$ and $0 \le r_v < v$     /* $k_v$ is the quotient and $r_v$ is the remainder when k-1 is divided by v */
8: x: = **c**                  /* store ciphertext **c** in intermediate value variable */
9: For j = 0 to **$r_v$**            /* special loop for the top window which may have fewer than v bits in it*/
10:    x:= x * x MOD **n**
11: End for
12: $w := \sum_{j=0}^{r_v-1} d_{vk_v} 2^j$      /* w is the value of the top window, which has $r_v$ bits in it */
13: Set $b_j$ := 0  for j = 0 to $r_v$ – 1.
14: $b_w$ = 1.
15: $r := \sum_{j=0}^{r_v-1} c_j b_j$         /* determine obscuring factor r = $c_w$ */
16: x := x * r MOD **n**                /* use obscuring factor */
17: For i = 1 to $k_v$             /* loop for all of the windows of v bits each */
18:    For j = 0 to **v**
19:          x:= x * x MOD **n**     /* perform the squares for this window */
20:    End for
21:    $w := \sum_{j=0}^{v-1} d_{v(k_v-i)} 2^j$     /* w is the value of i'th window */
22:    Set $b_j$ := 0  for j = 0 to v – 1.
23:    $b_w$ = 1.
24:    $r := \sum_{j=0}^{v-1} c_j b_j$        /* determine obscuring factor r = $c_w$ */
25:    x := x * r MOD **n**             /* use obscuring factor */
```

```
26: End For
27: m:= x     /* store plaintext m */
```

According to this embodiment, a conditional selection of the power of the message at a given iteration of the window method is replaced by the action of obtaining all powers of the message at each iteration, and then choosing the appropriate power to use for a given iteration.

When using the embodiment of the present invention shown in Table III as applied to the window method, each of the pre-computed powers of the message (e.g., **c**) may be multiplied by $\mathbf{s}^{(2^\wedge v - 1)}$ mod **n**. This is illustrated in Table V.

Table V

```
1: v: = the number of bits in the window.
2: Pick t at random MOD n          /* random number t */
3: s := $t^{-1}$ MOD n      /* determine second obscuring factor s */
4: $s_v = s^{2^\wedge v - 1}$ MOD n
5: $g_0 = 1$.
6: $c_0 = s_v$.
7: For i = 1 to $2^v$-1      /* loop through each integer less than $2^v$ to determine all $2^v$ obscuring factors*/
8:     $g_i = g_{i-1} * c$  MOD n
9:     $c_i = g_i * s_v$ MOD n   /* compute $c_i = c^i s^{2^\wedge v - 1}$ MOD n */
10: End for
11: k: = $\log_2$ n      /* k is the number of bits in d */
12: Compute $k_v$ and $r_v$ such that k-1 = v*$k_v$ + $r_v$ and $0 \le r_v < v$      /* $k_v$ is the quotient and $r_v$ is the remainder when k-1 is divided by v */
13: y := c * t MOD n          /* store ciphertext c in intermediate value y using t */
```

```
14: For j = 0 to **r**$_v$          /* special loop for the top window which may have fewer than v bits in it*/
15:     y:= y * y MOD **n**
16: End for
17: $w := \sum_{j=0}^{r_v-1} d_{vk_v} 2^j$     /* w is the value of the top window, which has $r_v$ bits in it */
18: Set $b_j$ := 0  for j = 0 to $r_v$ – 1.
19: $b_w$ = 1.
20: r = ($c_0$ AND $b_0$ ) OR ($c_1$ AND $b_1$ ) OR ... OR ($c_{rv-1}$ AND $b_{rv-1}$)
21: y := y * r MOD **n**              /* use obscuring factor */
22: $h = 2^v - 2^{r_v}$
23: y := y * $t^h$ MOD n      /* Correction factor needed for first window since it is less that v bits */
24: For i = 1 to $k_v$          /* loop for all of the windows of v bits each */
25:     For j = 0 to **v**
26:             y:= y * y MOD **n**     /* perform the squares for this window */
27:     End for
28:     $w := \sum_{j=0}^{v-1} d_{v(k_v-i)} 2^j$     /* w is the value of i'th window */
29:     Set $b_j$ := 0  for j = 0 to v – 1.
30:     $b_w$ = 1.
31:      r = ($c_0$ AND $b_0$ ) OR ($c_1$ AND $b_1$ ) OR ... OR ($c_{v-1}$ AND $b_{v-1}$)
32:     y := y * r MOD **n**              /* use obscuring factor */
33: End For
34: **m** := y * s MOD **n**  /* compute plaintext **m** using second obscuring factor */
```

Although the embodiments shown in Tables II, III, IV, and V illustrate decryption processing as an example, embodiments of the present invention may also be used to protect encryption processing as well. Further, embodiments of the present invention may be used in other cryptographic systems where exponentiation is used. For example, exponentiation is used in the well-known Diffie-Hellman key exchange, and also in the well-known Digital Signature Algorithm (DSA). In both of these algorithms, there is a modular exponentiation operation in which the exponent used is a randomly generated secret. Embodiments of the present invention may be used to deter attacks against these systems, and against any systems using a modular exponentiation operation.

Additionally, RSA may also be used for digital signatures. Embodiments of the present invention may be used for deterring attacks on digital signatures using RSA. In this scenario, a signer starts with a message **m** that he or she wants to sign. The signer transforms **m** into **h**, using a hashing and formatting protocol. The signer then computes the signature of **h** by $\mathbf{s} = \mathbf{h}^{\mathbf{d}} \bmod \mathbf{n}$. The signer sends the signature **s** and the message **m** to a verifier. The verifier transforms **m** into **h**, using the same hashing and formatting protocol. The verifier then computes $\mathbf{h'} = \mathbf{s}^{\mathbf{e}} \bmod \mathbf{n}$, and would accept the signature if $\mathbf{h} = \mathbf{h'}$. The modular exponentiations in the signing computations $\mathbf{s} = \mathbf{h}^{\mathbf{d}} \bmod \mathbf{n}$, and $\mathbf{h'} = \mathbf{s}^{\mathbf{e}} \bmod \mathbf{n}$, may also be obscured using the techniques disclosed herein.

Another embodiment of the invention can be used on an alternative exponentiation algorithm. The algorithm described in this invention processes the bits of the exponent starting with the high order bits. This is the most common implementation. However, an alternate exponentiation algorithm processes bits starting from the low order bits. The invention described here can be applied to that algorithm as well.

Embodiments of the present invention implement modular exponentiation for either encryption or decryption in the RSA algorithm (and may be applied to other cryptographic systems) in a secure manner even if an attacker has knowledge of what memory is accessed and what code branches (if any) are taken during execution of the algorithm. The invention thus provides better security than prior art implementations.

Although the following operations may be described as a sequential process, some of the operations may in fact be performed in parallel or concurrently. In addition, in some embodiments the order of the operations may be rearranged without departing from the spirit of the invention.

The techniques described herein are not limited to any particular hardware or software configuration; they may find applicability in any computing or processing environment. The techniques may be implemented in hardware, software, or a combination of the two. The techniques may be implemented in programs executing on programmable machines such as mobile or stationary computers, personal digital assistants, set top boxes, cellular telephones and pagers, and other electronic devices, that each include a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and one or more output devices. Program code is applied to the data entered using the input device to perform the functions described and to generate output information. The output information may be applied to one or more output devices. One of ordinary skill in the art may appreciate that the invention can be practiced with various computer system configurations, including multiprocessor systems, minicomputers, mainframe computers, and the like. The invention can also be practiced in distributed computing environments where tasks may be performed by remote processing devices that are linked through a communications network.

Each program may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. However, programs may be implemented in assembly or machine language, if desired. In any case, the language may be compiled or interpreted.

Program instructions may be used to cause a general-purpose or special-purpose processing system that is programmed with the instructions to perform the operations described herein. Alternatively, the operations may be performed by specific hardware components that contain hardwired logic for performing the operations, or by any combination of programmed computer components and custom hardware components. The methods described herein may be provided as a computer program product that may include a machine readable medium having stored thereon instructions that may be used to program a processing system or other electronic device to perform the methods. The term "machine readable medium" used herein shall include any medium that is capable of storing or encoding a sequence of instructions for execution by the machine and that cause the machine to perform any one of the methods described herein. The term "machine readable medium" shall accordingly include, but not be limited to, solid-state memories, optical and magnetic disks, and a carrier wave that encodes a data signal. Furthermore, it is common in the art to speak of software, in one form or another (e.g., program, procedure, process, application, module, logic, and so on) as taking an action or causing a result. Such expressions are merely a shorthand way of stating the execution of the software by a processing system cause the processor to perform an action of produce a result.

While this invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications of the illustrative embodiments, as well as other embodiments of the invention, which are apparent to persons skilled in the art to which the invention pertains are deemed to lie within the spirit and scope of the invention.